Methods

OBJECT-ORIENTED modeling and design is a systematic way to understand systems and build software applications. It is a *method* for software development. What is a method and what are its characteristics? This first group of articles addresses that broad question.

Many people equate methods with notation, "bubbles and arrows," but that is an unnecessarily limited view. *What is a Method* outlines the components in a complete method: concepts, notation, process, and patterns. No method yet incorporates all these elements fully, but this article presents a goal to strive for.

The Waterfall Model was characteristic of an earlier generation of "traditional" software development methods. (How quickly "traditions" evolve in the computer field!) That early method proposed a linear development path that required a large, perhaps unrealistic, amount of foresight. Real problems are difficult because they require iteration on several simultaneous dimensions. *Over the Waterfall and into the Whirlpool* describes six dimensions of iteration during development. This is an early article and the details of actually doing the iteration have been expanded in the next article in this collection.

A model changes its character as the development process continues through the life cycle. In *Layered Additive Models* I argue that the goal of a model is to capture design decisions as directly as possible, and the best way to do this is to evolve the model by adding elements, rather than by replacing them. A good modeling language is constructed as a set of layers that add successive details as the life cycle progresses. This kind of seamless development supports iterative design because different parts of the model can be at different life cycle stages yet coexist within a single model, in contrast to a development process that imposes rigid translation boundaries.

Some support tools allow an entire team to share a single workspace that contains the model being developed. Although the ability to share a single model is desirable, it is not sufficient. *A Private Workspace* shows that developers also need private workspaces in which they can work without being disturbed by constant changes from other developers. In this article I describe the parallel iterative development process that is most applicable to development of complex systems.

What Is a Method?

October 1995

Methods and Other Development Guides

There are a lot of books about analysis and design methods. There are also a lot of books about computer science principles, software engineering, and systems science. There are a lot of books telling you how to use various programming languages with various degrees of elegance. Recently there have been a number of books about patterns as a new approach to design. How does all of this stuff fit together and what is the role of a method?

A method is a generic guide to help people perform some activity. A method must apply to many different situations for many different people. There are methods for teaching skiing, doing accounting, or making sales. More particularly in the software business, we want methods for developing complex software. A method provides a framework for software development. But the term can't be so broad that it includes everything that one does during software development. It must be bounded. It is bounded on the generic side by the general principles of computer science and software engineering. It is bounded on the specific side by case-specific knowledge such as patterns, conventions, and domain experience. A method is the engineering practice in the middle.

What's in a Method

A method is a mixed bag of guidelines and rules, including the following components:

- A set of fundamental *modeling concepts* for capturing semantic knowledge about a problem and its solution. The modeling concepts are independent of how they are visualized. They are the inputs for semantic tools, such as code generators, semantic checkers, and traceability tools.

- A set of *views and notations* for presenting the underlying modeling information to human beings which allow them to examine and modify it. Normally the views are graphic, but multimedia interfaces are feasible with current technology. Graphic views use geometric arrangement and graphic markers to highlight portions of the semantic information. Usually each view shows only a part of the entire semantic model and different views may present the same semantic information in different forms.

- A step-by-step iterative *development process* for constructing models and implementations of them. The process may be described at various levels of detail, from the overall project management down to the specific steps to build low-level models. The process describes which models to construct and how to construct them. It may also specify measures of goodness for evaluating proposed designs.

- A collection of hints and rules-of-thumb for performing development. These are not organized into steps. They may be applied wherever they make sense. The concept of *patterns* is an attempt to describe case-based experience in a uniform way. Patterns represent specific design solutions to recurring problems. These may apply at various levels of detail, from large-scale architecture down to low-level data structures and algorithms.

Sometimes methods are criticized for being "just lines and bubbles." That is a simplistic view by people who just see a part of the picture (such as the notation). A balance between the different components is important. The test of a method is its ability to help people get correct results efficiently.

Modeling Concepts

The ultimate job of a method is to capture knowledge about a problem and build a solution to it. A model is a formal representation of a system at some level of abstraction. Models can be at different levels of abstraction. During requirements specification or analysis, the abstraction level may be high and implementation details may be omitted. During design and implementation, detail must be sufficient to actually construct a program.

Any model is built out of a collection of modeling concepts. Often people think of notations when they think of models, but underlying any notation is a set of logical modeling concepts that capture the semantics of the model. For example, important modeling concepts in OMT include classes, associations, generalizations, states, events, and other things. A usable method requires a "universal set" of modeling concepts—a sufficient set of concepts to model any system. Not all methods have a sufficient set of concepts. For example, many methods have relationships of aggregation and inheritance but no relationship of association; they are deficient and can't model things fully. A good method should have a small set of modeling concepts, but not so spare that it becomes tortuous to model common things. Set theory and most formal specification languages, although formal and universal, are too spare in practice for anyone but mathematical masochists.

The same model can be visualized in many different ways without changing its meaning. For example, a circuit diagram may be drawn using various electrical symbols, but the underlying meaning is the same—certain electrical components are connected together in particular ways. A circuit model can be subjected to various kinds of semantic analysis, such as computing current flows through the various components.

The modeling concepts in an object-oriented software development method include things such as classes, associations, inheritance, states, events, and functions. These all represent well-defined concepts with crisp definitions. They can be analyzed for correctness and for goodness of design. They can be mapped to and from code. They convey the meaning in the design. A code generator, for example, depends on the semantic model and doesn't care at all about the graphic notation.

The semantics of a model do not depend on the way it is drawn on a diagram. There are a lot of notations that use many different symbols for classes, associations, inheritance, and other modeling concepts. There is a wide variation in the appearance of the notations. Under the surface, however, the modeling concepts from most of these methods are fairly similar. Most methods mean the same thing by a class, for example. The superficial differences in notation can obscure the deeper similarities in notation. Grady Booch and I began

working to unify our two methods after we discovered that the fundamental modeling concepts were almost identical, although the surface symbols were different.

Metamodels

A metamodel is a model that describes other models. A metamodel for a method model describes the concepts in the method and their relationship to each other. It defines and restricts how atomic concepts can be connected to form complex constructs. It defines the legal models that can be constructed within the method. It describes the information that must be captured by computer-aided software engineering (CASE) tools to support the method.

Because methods have both modeling concepts and notation, a metamodel for the method needs to describe both. The metamodel should first describe the underlying semantic model. Then it should describe the mapping between the semantic model and the various views. Usually the views closely mirror the semantic model, so it may be unnecessary to actually draw the metamodel for the views. Instead the metamodel can describe the graphical syntax of diagrams and explain how to map diagrams into logical models.

Because a metamodel is itself a model, a method can describe itself. This can be a bit confusing for most people, because the same concepts appear in the metamodel more than once at different semantic levels: we can describe a modeling concept using itself and other modeling concepts, which can sometimes make the head spin.

Notation

People need notation to construct, examine, and manipulate models. People can't interact directly with a logical model. Any model has to be represented somehow, if only by a text description or table. Most methods provide a graphical visualization of most of the key models. Various shapes, lines, arrangements of symbols, and graphic marks represent the underlying semantic information. This is what people see first about most methods, and they can get excessively attached to a particular notation which is intended after all to be a pipeline to the underlying concepts. The same model can be drawn in many different ways using different symbols, even though they all mean the same thing. Whether classes are drawn as rectangles, clouds, ellipses, or other shapes, they all have the same meaning. Don't lose the model for the icons!

A diagram is more than a simple mapping from a model. A diagram adds information, but it is not semantic information. All the semantic information comes from the semantic model. A diagram selects, organizes, and displays information to highlight things of interest and suppress less important things. A diagram is a *projection* from a model; it doesn't have to include everything. It may require several diagrams to show all the information in a model. This is all right; you don't need to see it all at once, and if you did, it would be too confusing.

Text formats are views too. A language, a table, a set of records are all ways of presenting semantic information to people. The same kinds of model-view mappings can be used to handle graphic and text views.

The geometric organization of a diagram adds an aesthetic component to the model. The geometric arrangement of symbols can be used to emphasize things that are closely related, but the criteria for affinity are left up to the person. Graphic markers, such as density, color, texture, fill patterns, multiple lines, text fonts, size, etc., can be used to convey various kinds of information for quick apprehension by humans. The trick is to avoid trying to show too much at one time.

Different people may adopt different conventions to format their diagrams, just as programmers may follow different indentation rules in writing C++ code. For example, the use of orthogonal lines in OMT diagrams is just a formatting convention. A convention is a way for an individual to tailor a notation to individual taste. It is not part of the method itself, but I think of it as part of the penumbra (shadow) of the method.

Artifacts

Models and diagrams are the artifacts of the development process. These are the documents that developers, domain experts, managers, and customers can examine, argue about, and criticize. These are the deliverables that measure progress. These are the artifacts that tools must produce. These are the files that can be exchanged among people and among tools. The final system is just a very complete model that includes code among its components.

Artifacts are what are exchanged, so artifacts are the things that need to be standardized. It is not so important to standardize the development process itself, because different people can get the same results in different, equally valid ways, but the results themselves must be represented in some well-defined format so that people can understand them and tools, including CASE tools as well as compilers and GUIs (graphical user interfaces), can

manipulate them. Not only the content of the artifacts but their physical format must be standardized to permit interchange among tools supporting a single method.

However, any attempt to standardize across many different methods is a hopeless idea, similar to trying to standardize a common language from C++, Smalltalk, and Lisp. One of the most important aspects of any method is the choices it makes about which concepts are important and which can be omitted. It is balance that makes a successful method, not a checklist of features. The idea of a common core method is misguided, not because it is undesirable, but because it is impossible. Any "core" method is inevitably yet another method, with its own choice of concepts, notation, and priorities, subject to the same evaluation criteria as any other new method. There is no evading the necessity to make choices.

Process

Artifacts (models and diagrams) are target points on the development map; the process is the path taken to get to them. Like routes on a roadmap, there are many possible processes that produce the same results. The choice between them may depend on personal taste as much as whether one is "better."

A process is a guide telling how to produce a model. It provides a framework for development, describing the artifacts to be produced and the steps for producing them. At the top level, a process describes the development lifecycle and the iteration steps within it. At a lower level, a process provides a framework for producing models: steps to construct the model, guidelines for discovering components of it, design principles to be followed, measures of goodness, cross-checking and consistency rules, and red flags for possible problems.

You can't expect a method to tell you everything to do. Writing software is a creative process, like painting or writing or architecture. There are principles of painting, for example, that give guidelines on composition, color selection, and perspective, but they won't make you a Picasso. You still have to select your subject and decide on your approach. Software development is the same. You can't follow a recipe without having to think. Some methods claim to fully automate the process, to tell you every step to follow so that software design is painless and faultless. They are wrong. It can't be done. What can be done is to supply a framework that tells how to go about it and that identifies the places where creativity is needed. But you still have to supply the creativity. A process can give guidelines for identifying objects, for example, but you still have to use your judgment to select the objects from the problem.

A process can operate at several levels: the high-level management process, which describes the course of the entire project, focusing on deliverables and task planning; the middle level technical macroprocess, which describes the software lifecycle and the iterations needed to produce a design; and the lower level technical microprocess, which describes the actions needed to construct pieces of the detailed design. The upper levels can be prescriptive enough, outlining a step-by-step plan for development; but the steps are large and somewhat fuzzy. At the lower levels it is much less realistic to expect a rigid step-by-step process. Instead what can be offered is a set of guidelines to be applied and rules for when to apply them. The sequencing of the guidelines is usually fairly open.

In my work on OMT I have outlined two levels of technical process. I have left the management process for others. There is more scope for tailoring the process than the notation. If you want to do things differently than I recommend, then go ahead, provided you don't blame me if things go wrong.

Patterns

Over time many people have observed that there are good solutions to certain problems that come up repeatedly in good designs. Over the years in any creative craft, the practitioners learn good ways to solve certain kinds of problems, and the good solutions tend to be reused. Rather than create a brand new design from first principles each time, expert designers save a lot of work by incorporating these "canned solutions" into their own designs. House builders have conventional solutions to building windows, running plumbing, or laying out kitchens. These standard solutions may not be optimal in every respect, but they are good, serviceable designs that have been validated by years of experience. These "solutions waiting for problems" are called *patterns*. Architects, tailors, cabinetmakers, and other craftsmen have traditionally kept pattern books that showed good solutions to many design problems. Novice craftsmen learn from the experts by examining their patterns.

Software people have had a great handicap in learning to design. There are no "program museums" where you can examine great programs to emulate. At best, novices can undergo an apprenticeship by working with experts and learning their tricks. At worst, new designers are on their own, and the results often show it.

Methods have helped by supplying a framework and general principles for development, but what has been missing is the case-based experiential knowledge that exists in engineering, for example. Recently the Patterns Movement has sprung up to fill this gap. People have begun collecting, cataloging and explaining useful patterns so that other designers

can learn from them. Books such as *Design Patterns* by Gamma, Helm, Johnson, and Vlissides [1] contain lists of low-level design patterns. These patterns are the heart of the detailed design process. They are the proven solutions to frequently-encountered problems.

Rules of Thumb

Many method books have contained rules of thumb for building designs. But what is a rule of thumb but a pattern in a less formal presentation? So software patterns have actually been around for a long time. Does this mean that the Patterns Movement is just a gimmick? Not at all, any more than the Object Movement is a gimmick. There is virtue in identifying a general approach to capturing and representing knowledge, and patterns are a good way to describe this kind of knowledge.

I find that I can recast some of my own advice as patterns. For example, I recommended trying to reify (turn into an object) operations that can have variant implementations or algorithms so that the implementations are easy to change without having to change client code. This is described in *Design Patterns* as the Strategy pattern.

I have other rules that do not appear in the *Design Patterns* book, but they could be stated as patterns. For example, a pattern to add redundant associations to optimize frequently traversed paths could be called "Redundant."

Other rules of thumb are not so easily cast as patterns, however. Advice to "keep methods small" seems best expressed as a rule. Perhaps there is room for both patterns and more general rules without specific implementations (or perhaps I just haven't seen how to write them as patterns).

Process and Patterns

What's the relationship between process and patterns? Should a method contain patterns or are they separate?

I would distinguish three kinds of how-to-do-it components for methods: process, general design rules, and patterns. A process provides a time-sequence-oriented framework for work. It may not be rigidly ordered, but it is normally procedural, with advice of the form "do this, then do that, until such and such is true." A process is a kind of recipe to follow, although the steps may be complex and some of them may require application of patterns.